<System On Chip>

- UART -

· 제출 레포트

전자공학과

2015142011 류찬

목 차

UART (RS-232C)

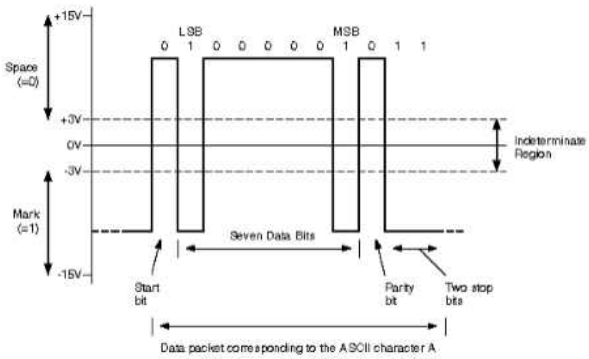
- Universal Asynchronous Receiver and Transmitter

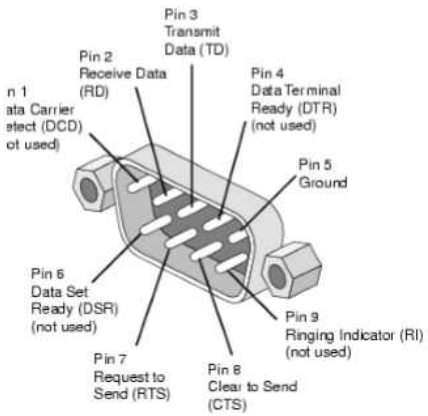
- 비동기 전이중(full-duple) 1:1 통신방식
-> 하나의 선으로 수신과 통신 모두를 하는 반이중 방식과는 반대로 전이중 방식이란 두 개의 Line이 각자 tx, rx의 역할을 하는 것이다.

- 최대 통신거리 : 1 5 m

- 최고 통신속도 : 20kb/s, BaudRate에 따라 구분 가능

- 최대 출력전압 : (+-)25V, 최대 입력전압 : (+-)15V





UART Protocol

- Baud_CLK는 통신속도에 따라 변경
-> 보드레이트에 따라 통신속도가 결정되는데, 이에 맞춰 보드 클락 속도를 설정한다.

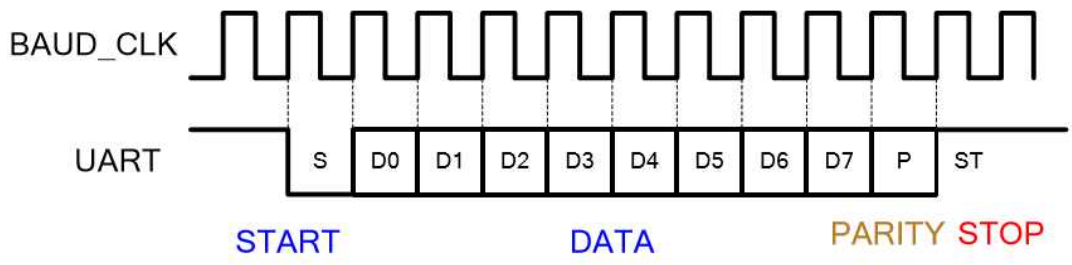
- UART는 Default High('1')
-> UART는 따로 설정하지 않으면 default값이 '1'로 설정되어 있다.

- START(1b) : HIGH -> LOW (falling_edge)
-> 밑의 그림에서 UART가 '1'에서 '0'으로 넘어갈 때 START로 상태가 이동한다.

- DATA(8b) : LSB First
-> 데이터의 최하위 비트(LSB)가 처음에 온다.

- PARITY : Odd 혹은 Even으로 설정 가능
-> 패리티 비트는 '1' 혹은 '0' 모두 설정이 가능하다.

- STOP : High
-> 패리티 비트가 나온 후 High시그널이 오면 상태가 STOP으로 변경된다.



DE2 Board의 UART 구성

- MAX232 : TTL ↔ UART Level 변환 RS-232 Transceiver

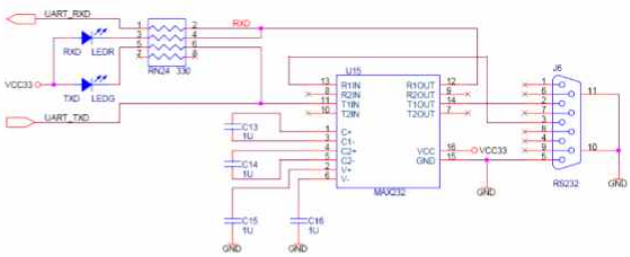


Figure 4.16. MAX232 (RS-232) chip schematic.

Signal Name	FPGA Pin No.	Description
UART_RXD	PIN_C25	UART Receiver
UART_TXD	PIN_B25	UART Transmitter

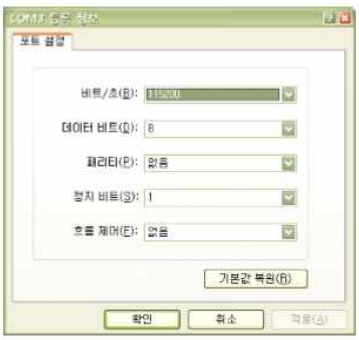
Table 4.10. RS-232 pin assignments.

-> 다음은 완성된 UART의 MAX232의 구성도이다.
밑의 UART_RXD와UART_RXD는 이후 Pin Assignment단계에서 필요하기에 기억해두면 좋다.

Baud_Rate

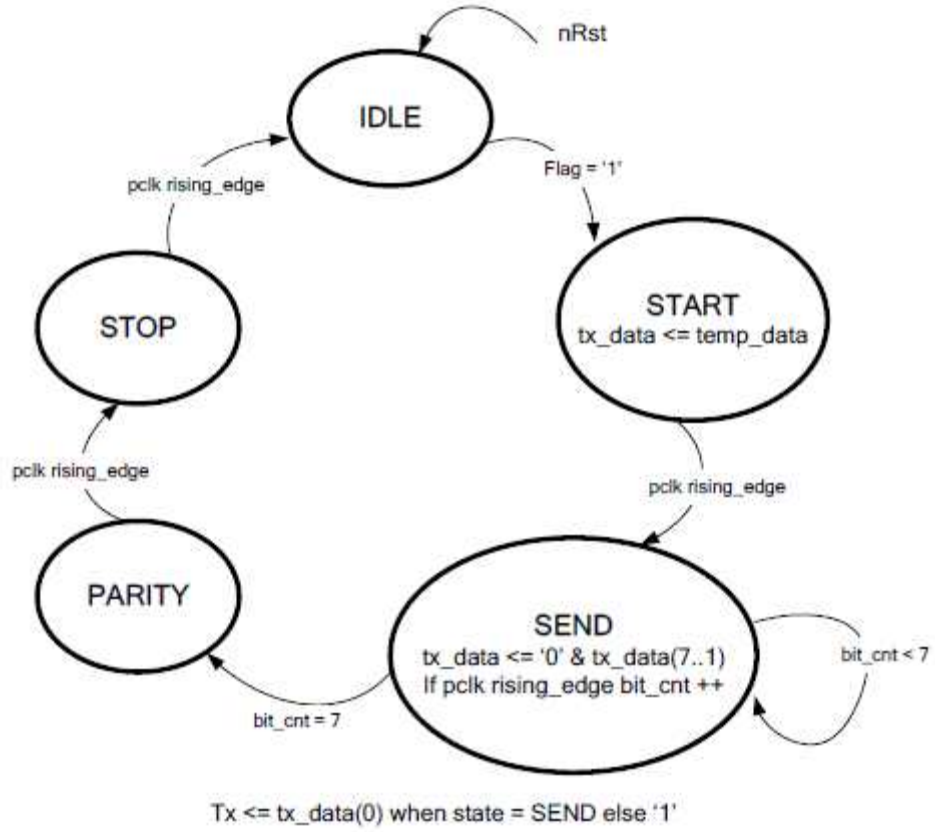
- UART의 초당통신속도
- 9600bps ~ 921600bps
- 일반적으로 9600bps, 115200bps를 많이 사용
- 우리 실습에서는 115200bps만을 사용
- 115200bps : 8.68055us

-> 1/115200 = 약 0.0000086805556 => 8.68055us으로 설정한다.



이후 보드 실습에서 hyperterminal에서 설정은 다음 사진과 같이 하면 된다.

UART_TX 설계



위는 transmitter의 DIagram이다.
5가지의 state로 구성되는데, IDLE -> START -> SEND -> PARITY -> STOP으로 진행된다.
1) 처음 nRst = '0'에서 state는 IDLE로 시작하게 된다. 이는 @@@@@@@@@@@@@@@@@@
2) Flag가 '1'로 설정된 상태라면 state는 IDLE에서 START로 이동하게 된다.
3) pclk rising_edge상태에서 state는 START에서 SEND로 이동한다.
4) SEND상태에서 tx_data는 7번째 데이터부터 1번 데이터까지 하나씩 0과 '&'연산하여 tx_data에 넘겨준다. 이는 shift를 하며 넘겨주는 것과 같다.
5) bit_cnt가 '7'이 되면 state는 SEND에서 PARITY로 이동하게 된다. PARITY는 사용자의 용의에 따라 ODD or EVEN중에 선택할 수 있다.
6) PARITY상태에서 pclk가 rising_edge이면 state는 STOP으로 이동한다.
7) STOP상태에서 pclk가 rising_edge이면 state는 다시 IDLE로 돌아간다.

모든 상태머신의 데이터 판정은 pclk가 rising_edge일 때 이루어진다.
nRst은 처음 IDLE로 이동할 때 '0'을 판정하고 이후로는 항상 '1'일 때 state의 변화를 감지한다.

Code of UART_TX

```
1   library ieee;
2      use ieee.std_logic_1164.all;
3      use ieee.std_logic_arith.all;
4      use ieee.std_logic_unsigned.all;
5
6   entity uart_tx is
7      port(
8         nRst      : in std_logic;
9         clk       : in std_logic;
10        start_sig : in std_logic;
11        data      : in std_logic_vector(7 downto 0);
12        tx        : out std_logic;
13        busy      : out std_logic
14     );
15  end uart_tx;
```

TX설계에 앞서 먼저 library와 entity를 설정해 준다.
library 문을 통해 코드 내에서의 연산에 필요한 요소들을 가능하게 해준다.
entity선언에서 input과 output의 코드를 작성해준다.

```
17  architecture BEH of uart_tx is
18
19     type   state_type is (IDLE, START, SEND, PARITY, STOP);
20     signal state     : state_type;
21     signal cnt       : std_logic_vector(8 downto 0);
22     signal pclk      : std_logic;
23     signal start_d   : std_logic;
24     signal flag      : std_logic;
25     signal temp_data : std_logic_vector(7 downto 0);
26     signal tx_data   : std_logic_vector(7 downto 0);
27     signal bit_cnt   : std_logic_vector(3 downto 0);
```

본격적인 코드를 진행하기에 앞서 포트 내부에서 동작할 각각의 signal을 설정해준다.
signal의 선언은 이전의 DIagram에서 필요한 요소를 파악한 뒤 선언해 주면 된다.
각각의 signal은 1bit의 값을 가지기 때문에 요구되는 bit수에 맞추어 vector값으로 설정 할 수 있다.

```
29  begin
30
31      process(nRst, clk)
32      begin
33          if(nRst = '0') then
34              start_d   <= '0';
35              flag      <= '0';
36              temp_data <= (others => '0');
37          elsif rising_edge(clk) then
38              start_d <= start_sig;
39               if(start_d = '0')and(start_sig = '1') then
40                  flag      <= '1';
41                  temp_data <= data;
42              elsif(state = START) then
43                  flag <= '0';
44              end if;
45          end if;
46      end process;
```

본격적인 architecture내의 process문 진행이다.
먼저 nRst, clk신호에 맞추어 데이터를 판별할 수 있다.
처음으로 nRst = '0'이 될 때 start_d, flag, temp_data의 전체 값을 '0'으로 초기화시킨다.
이후 state의 판별에서는 nRst의 값이 '1'일 때에 진행하게 된다.
다음부터의 process에서 nRst의 값은 default '1'이라고 생각하면 편하다.
clk신호가 rising_edge일 때 start_d신호에 start_sig의 값을 저장한다.
이는 start_d신호가 start_sig의 값이 clk에 맞추어 delay된 신호라는 것을 의미한다.
if문에서 (start_d='0' and start_sig='1')이면 flag에 '1'을 세트하고 temp_data에 data값을 저장시킨다.
이는 start_sig의 값은 0 -> 1로 바뀌고, clk타이밍에 start_d신호는 아직 0에서 변하지 않은 것을 이용하여 start_sig가 0에서 1로 바뀌는 순간, 즉 일종의 rising_edge를 구현하려는 목적임을 짐작할 수 있다.
이는 위 if문의 조건을 만족하였을 때가 원하는 타이밍이 아닐 가능성을 고려하여 flag의 값을 세트하는 것이다.
state가 START로 이동하면 flag는 '0'으로 초기화된다.

```
48      process(nRst, clk)
49      begin
50         if(nRst = '0') then
51            cnt <= (others => '0');
52            pclk <= '0';
53         elsif rising_edge(clk) then
54            if(cnt = 433) then -- 115200bps
55               cnt  <= (others => '0');
56               pclk <= not pclk;
57            else
58               cnt <= cnt + 1;
59            end if;
60         end if;
61      end process;
```

처음 Baud_Rate의 값에 맞추어 약 8.68us의 속도로 통신하게 된다.
그것을 위해 cnt는 0부터 433으로의 증가 즉, 434개의 변위을 갖게 되고 이에 맞추어 pclk는 0, 1, 0으로 not연산을 통해 바뀌게 된다.
즉 pclk가 0에서 다시 0으로 돌아올 때 cnt는 868번 이동한다.
1bit는 0.01us의 통신속도를 가져서 이를 통해 8.68us의 구현을 할 수 있다.

```
63      process(nRst, pclk)
64      begin
65         if(nRst = '0') then
66            state   <= IDLE;
67            bit_cnt <= (others => '0');
68            tx_data <= (others => '0');
69            busy    <= '1';
70         elsif rising_edge(pclk) then
71            case state is
```

다음으로 process는 이전 process에서 설정해 둔 pclk의 rising_edge를 통하여 진행한다.
여기서 nRst = '0'이면 state가 IDLE이 되며 bit_cnt와 tx_Data를 초기화하고 busy를 '1' 로 세트시킨다.
이후 case문을 통해 process문이 진행된다.
각각의 state상황은 nRst = '1'이며 pclk의 rising_edge일 때에 판별한다.

```
73              when IDLE =>
74                 if(flag = '1') then
75                    state <= START;
76                 else
77                    state <= IDLE;
78                 end if;
79
80                 bit_cnt <= (others => '0');
81                 tx_data <= (others => '0');
82                 busy    <= '1';
83
84              when START =>
85                 state   <= SEND;
86                 tx_data <= temp_data;
87                 busy    <= '0';
88
89              when SEND =>
90                 if(bit_cnt = 7) then
91                    bit_cnt <= (others => '0');
92                    state   <= PARITY;
93                 else
94                    bit_cnt <= bit_cnt + 1;
95                    tx_data <= '0' & tx_data(7 downto 1);
96                 end if;
97
98              when PARITY =>
99                 state <= STOP;
100
101             when STOP =>
102                state <= IDLE;
103                busy  <= '1';
104
105             when others =>
106                state <= IDLE;
107          end case;
108       end if;
109    end process;
```

nRst = '0'일 때 state는 IDLE상태에 돌입하고, 이후 IDLE state의 진행은 nRst가 '1'의 값을 계속해서 가질 때이다.
이전에 설정한 flag가 '1'일 때 state는 IDLE에서 START로 이동하게 된다.
그렇지 않으면 state는 계속해서 IDLE에 위치하게 된다.
IDLE상태에서 bit_cnt, tx_data는 0으로 초기화되고, busy신호는 '1'을 유지한다.

이후 START신호에 들어가면 tx_data에 temp_data의 값을 넣고 busy신호를 '0'으로 초기화 시킨다. 이는 데이터 송신 중임을 의미한다.
그리고 state는 SEND상태가 되어 다음 pclk rising_edge에 맞춰 SEND case에 들어간다.

SEND상태에서 data를 전송한다.
데이터는 LSB방식으로 8bit를 전송하기 때문에 bit_cnt는 7로 설정한다. 7 downto 1 즉 최하위 비트를 제외한 데이터를 보낼 수 있을 것이다.
7번 비트부터 1번 비트까지를 &연산하여 '0'과 더해서 tx_data에 할당시킨다.
이를 bit_cnt가 7이 될 때까지 반복하다 bit_cnt가 7이 되면 state는 PARITY로 이동하고 bit_cnt는 다시 '0'으로 초기화된다.

이후 PARITY상태와 STOP상태일 때 pclk의 rising_edge에 맞춰 state는 진행된다.
STOP상태에서는 busy를 다시 '1'로 세트하여 데이터 통신이 완료되었다는 것을 보여준다.
다른 상황에서 state는 IDLE상태이다.
이후 process문과 내부의 조건들을 모두 종료시킨다.

```
111      tx <= tx_data(0) when state =  SEND else
112            '0'        when state = START or state = PARITY else
113            '1';
114
115  end BEH;
```

이후 out port인 tx에 state가 SEND상태일 때에 tx_data LSB비트를 저장시킨다.
state가 START, PARITY상태일 때에는(이 사이에 SEND state가 위치한다) state는 '0'상태이다.
다른 state에서 tx는 '1'을 저장받는다.

이것으로 UART-tx의 설계를 마친다.

Test_Bench of UART_tx

```
1   library ieee;
2      use ieee.std_logic_1164.all;
3      use ieee.std_logic_arith.all;
4      use ieee.std_logic_unsigned.all;
5
6   entity tb_uart is end;
7
8   architecture BEH of tb_uart is
9
10     component uart_tx is
11        port(
12           nRst      : in std_logic;
13           clk       : in std_logic;
14           start_sig : in std_logic;
15           data      : in std_logic_vector(7 downto 0);
16           tx        : out std_logic;
17           busy      : out std_logic
18        );
19     end component;
20
21     signal nRst      : std_logic;
22     signal clk       : std_logic;
23     signal start_sig : std_logic;
24     signal data      : std_logic_vector(7 downto 0);
25     signal tx        : std_logic;
26     signal busy      : std_logic;
27     signal int_cnt   : std_logic_vector(80 downto 0);
```

test bench에서 tx의 entity 요소들을 component로 선언 후 signal로 설정한다.
int_cnt는 내부의 타이밍 설정을 위해 signal로 선언한다.

```
29  begin
30
31      process
32      begin
33          if(NOW = 0 ns) then
34              nRst <= '0', '1' after 200 ns;
35          end if;
36          wait for 1 sec;
37      end process;
38
39      process
40      begin
41          clk <= '0', '1' after 5 ns;
42          wait for 10 ns;
43      end process;
44
45      process(nRst, clk)
46      begin
47          if(nRst = '0') then
48              int_cnt <= (others => '0');
49          elsif rising_edge(clk) then
50              int_cnt <= int_cnt + 1;
51          end if;
52      end process;
53
54      start_sig <= '1' when int_cnt = 1000 else
55                   '0';
56      data      <= x"A7" when int_cnt > 900 and int_cnt < 1400 else
57                   x"00";
```

구성 요소의 선언이 끝났으면 이후 process문을 진행한다.

nRst은 '0'으로 설정되어 있다가 200ns이후부터는 '1'로 할당한 후 그 값을 유지한다.
여기서 1sec동안 유지하는데, 이 이유는 modelsim의 시뮬레이션에서 1sec의 시간은 매우 길고, 그 값으로 임의로 설정하여 시뮬레이션동안은 계속하여 '1'을 유지하게 만든다.

다음 process에서 clk는 5ns를 기준으로 '0' -> '1' -> '0'의 패턴을 반복한다.
즉 10ns주기의 '0', '1'의 파형이 생성된다.

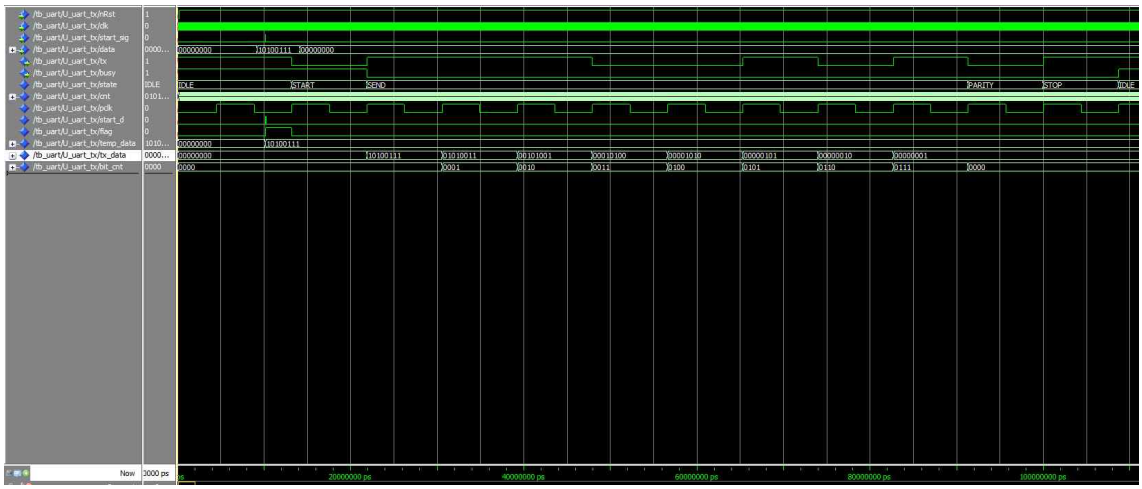
마지막 process에서는 nRst의 값이 '0'일 때 int_cnt의 값을 초기화한다.
그리고 nRst이 '1'일 때에(즉 200ns이후부터 1sec<-시뮬레이션이 끝날 때 까지)는 clk의 rising_edge일 때 int_cnt의 값을 1씩 증가시킨다.

다음에 이전 process에서 바뀌는 int_cnt의 값을 기준으로 start_sig의 값을 바꾼다.
이 시뮬레이션에서는 임의로 int_cnt가 1000일 때 '1'로 만들었다. 다른 값일 때에는 '0'이다.
data는 int_cnt가 900~1400일 때에는 x"A7", 다른 값에서는 x"00"을 주기로 했다.

```
59      U_uart_tx : uart_tx
60      port map(
61          nRst      => nRst,
62          clk       => clk,
63          start_sig => start_sig,
64          data      => data,
65          tx        => tx,
66          busy      => busy
67      );
68
69  end BEH;
```

이후 라벨을 지정하여 포트맵을 설정해 주고 TB를 종료시킨다.

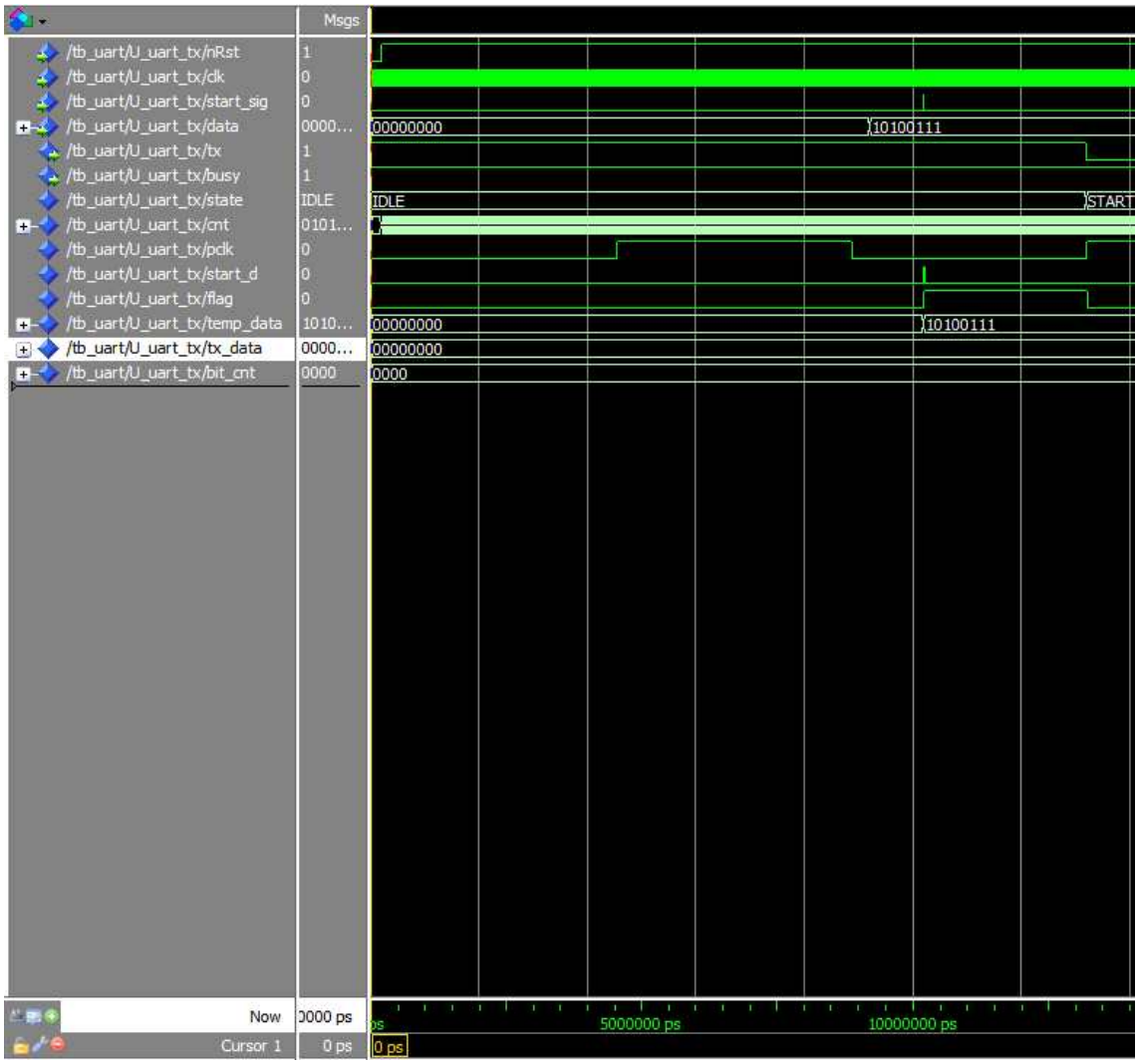
Simulation of UART_tx



다음 시뮬레이션은 UART의 tx를 test bench를 통해 modelsim 내에서 구현한 것이다.

여기서 nRst의 값은 처음 200ns까지만 '0'이고 이후 state가 IDLE에서 시작하여 다시 STOP을 지나 IDLE로 돌아오는 동안 계속해서 '1'로 유지됨을 확인 가능하다.

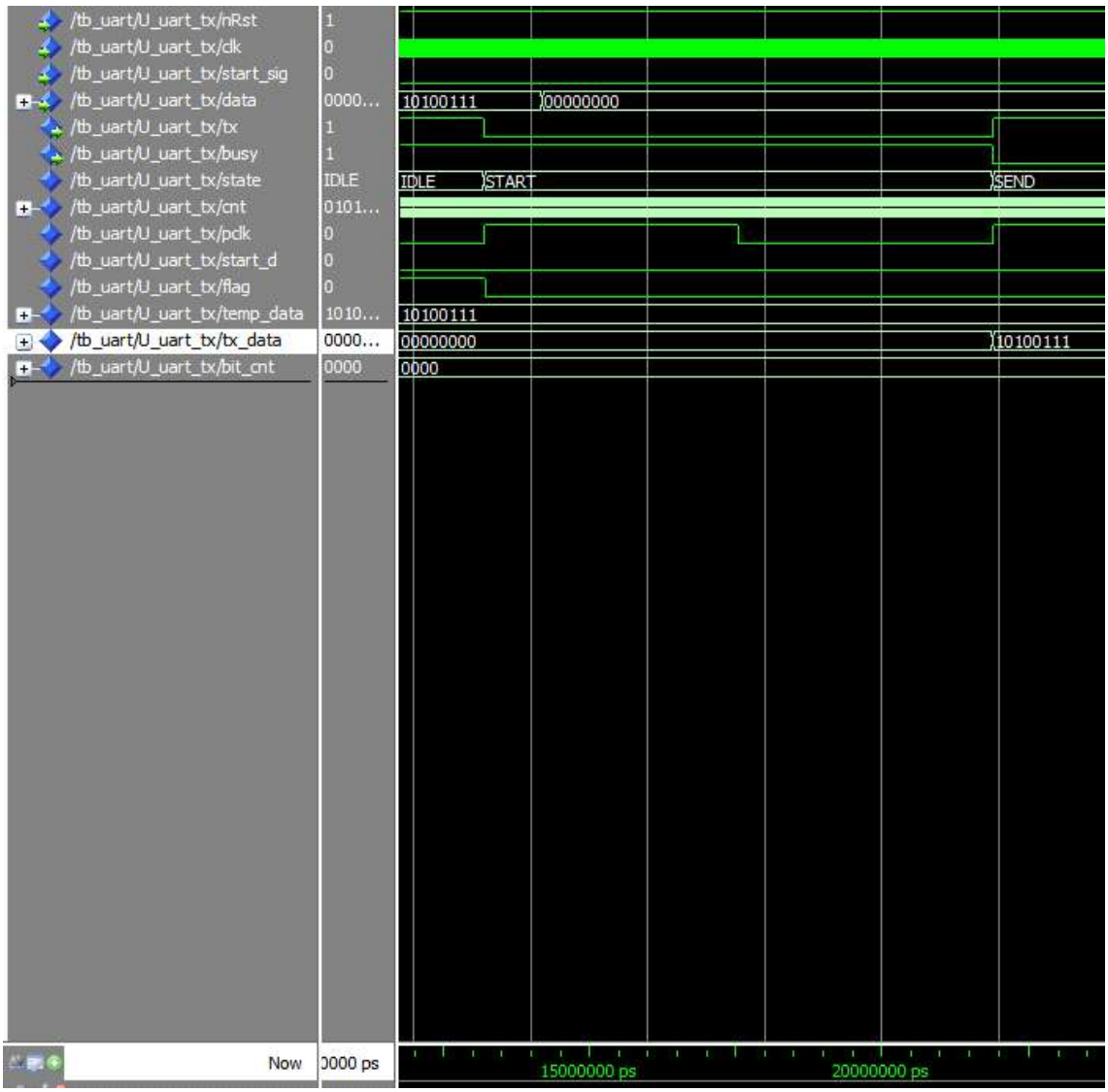
이후로 각 state별로 나누어 시뮬레이션의 분석을 하도록 하겠다.



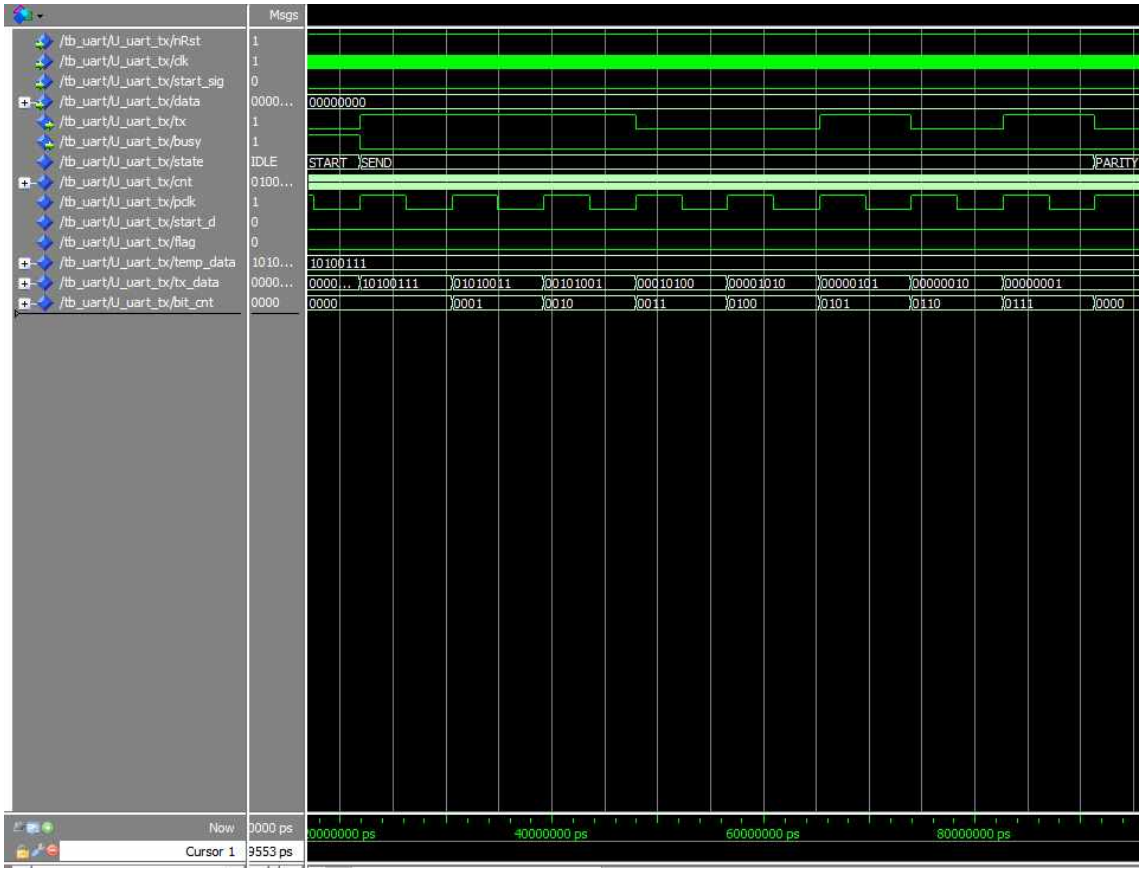
nRst이 '0'일 때 temp_data, flag, start_d모두가 '0'으로 초기화된다.
원래 값이 초기화되어 있을 수 있지만 안정성을 위해 코드에서 초기화 하였다.

data의 값은 int_cnt가 900일 때에 0xA7 즉 0b10100111으로 할당된다.
또 start_d신호는 '0', start신호는 '1'인 순간 flag는 '1'로 세트된다.
이 때 temp_data는 data의 값을 할당받는다.
(clk는 시뮬레이션 내에서 빠른 속도로 진행된다)

flag는 '1', pclk가 rising_edge일 때 state는 START로 이동하고 다시 flag는 '0'으로 초기화 된다.

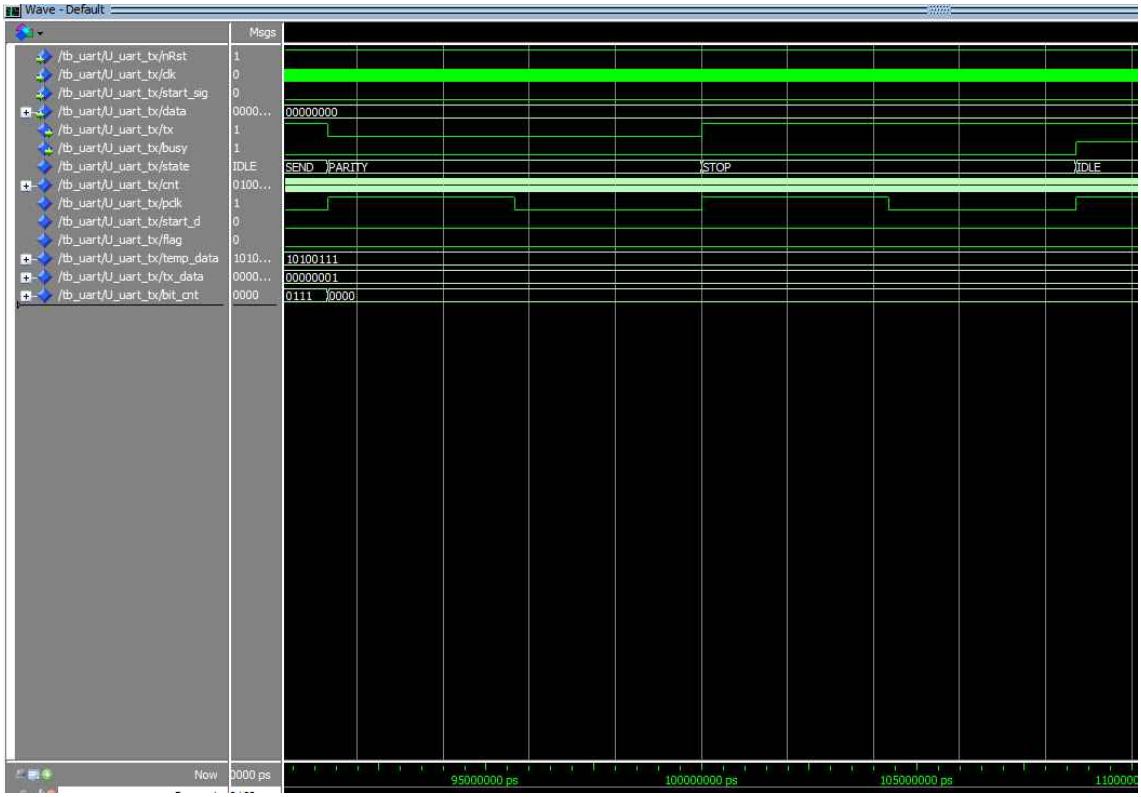


data는 int_cnt가 1400일 때 다시 0x00으로 초기화된다.
START상태에서 pclk가 rising_edge이면 SEND상태로 이동하게 된다.



SEND상태에서 pclk의 값이 8번 반복될 때까지 state는 유지된다.
그 값은 bit_cnt의 값이 '7' 즉 0b00000111이 될 때까지 pclk의 rising_edge를 따라 증가하며 그 값에 도달한 이후는 PARITY로 이동하고 다시 '0'으로 초기화되는 것을 보인다.

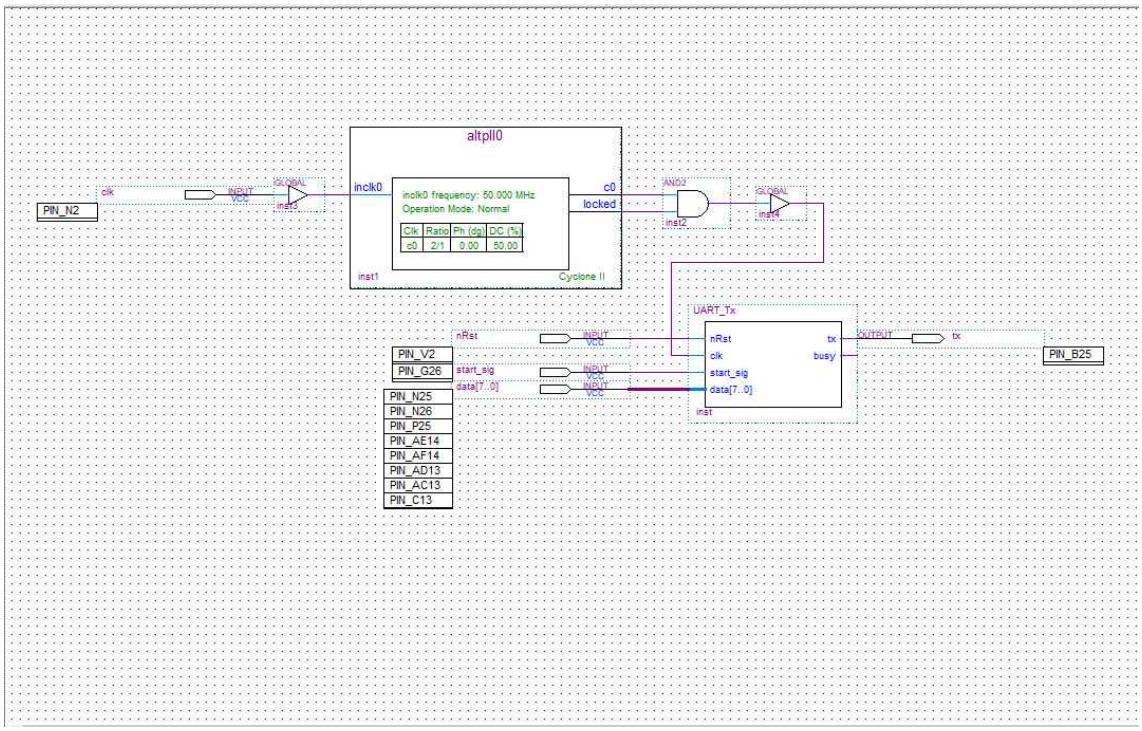
pclk의 rising_edge마다 tx_data의 값은 0과 '&'연산되어 다시 저장된다. 이는 값이 왼쪽에서 오른쪽으로 shift연산되는 결과를 보인다.



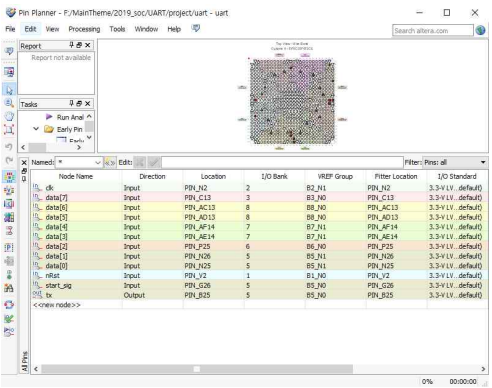
각 PARTIY, STOP사태에서 pclk가 rising_edge일 때 다음 state로 이동하게 된다.

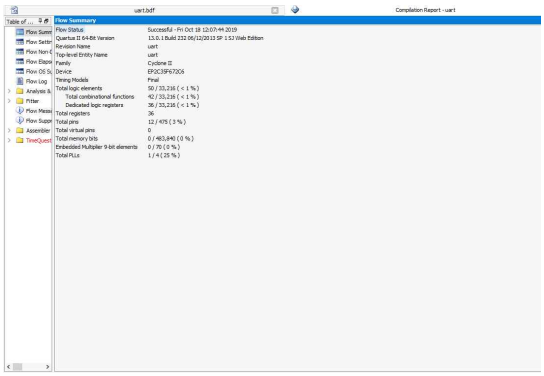
STOP비트가 끝날 때에 busy신호는 다시 '1'로 세트되어, 데이터의 전송이 끝났음을 보여준다.

이렇게 만든 tx를 quartus와 hyperterminal을 통해 확인해 본다.



quartus에서 UART_tx를 symbolize 한 뒤 quartus의 block diagram을 구성한다.

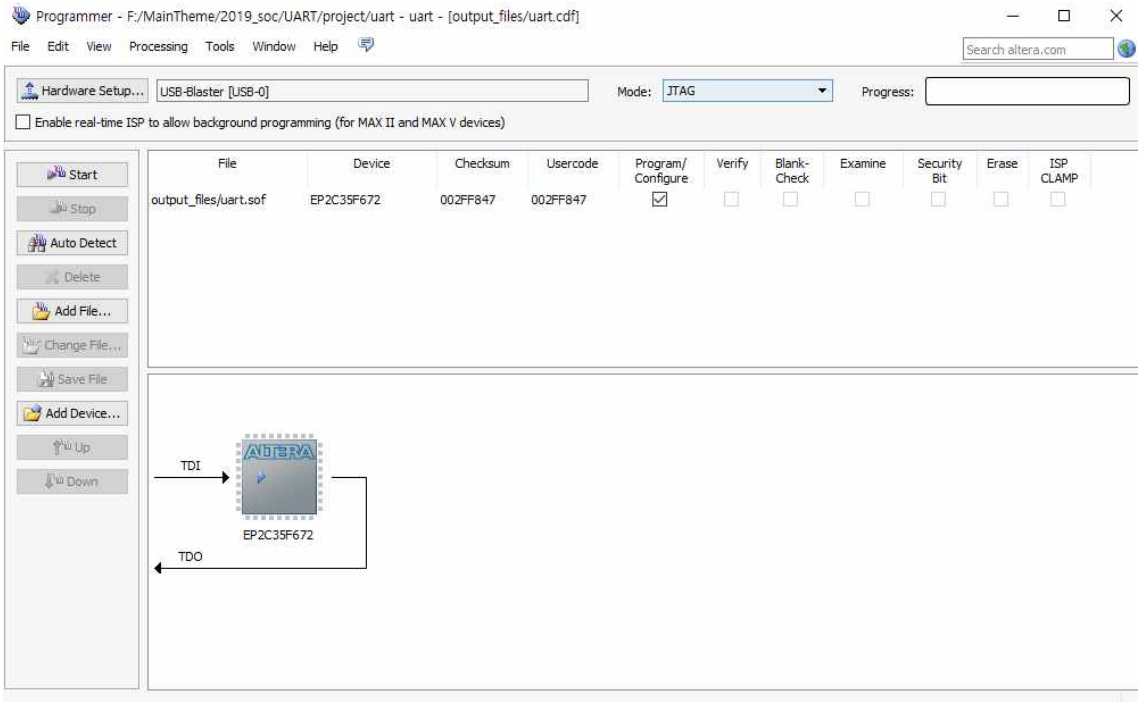




PIN설정을 한 이후 시뮬레이션 Error을 확인한다.

chip data를 설정한다. 이 때 PIN설정은 교수님이 첨부해주신 데이터를 참고하여 할 수 있다.

tx 부분의 chip data는 이전에 말해두었듯 PIN_B25에 연결시킨다.

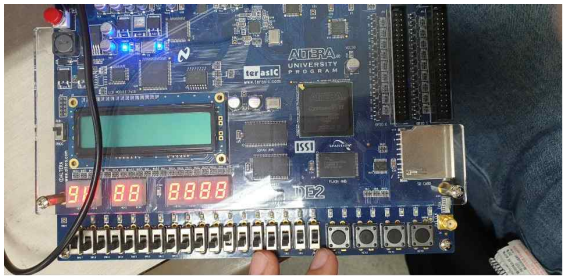


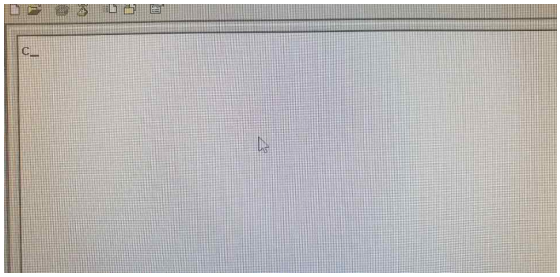
65	0x41	A	97	0x61	a
66	0x42	B	98	0x62	b
67	0x43	C	99	0x63	c
68	0x44	D	100	0x64	d
69	0x45	E	101	0x65	e
70	0x46	F	102	0x66	f
71	0x47	G	103	0x67	g
72	0x48	H	104	0x68	h
73	0x49	I	105	0x69	i
74	0x4A	J	106	0x6A	j
75	0x4B	K	107	0x6B	k
76	0x4C	L	108	0x6C	l
77	0x4D	M	109	0x6D	m
78	0x4E	N	110	0x6E	n
79	0x4F	O	111	0x6F	o
80	0x50	P	112	0x70	p
81	0x51	Q	113	0x71	q
82	0x52	R	114	0x72	r
83	0x53	S	115	0x73	s
84	0x54	T	116	0x74	t
85	0x55	U	117	0x75	u
86	0x56	V	118	0x76	v
87	0x57	W	119	0x77	w
88	0x58	X	120	0x78	x
89	0x59	Y	121	0x79	y
90	0x5A	Z	122	0x7A	z

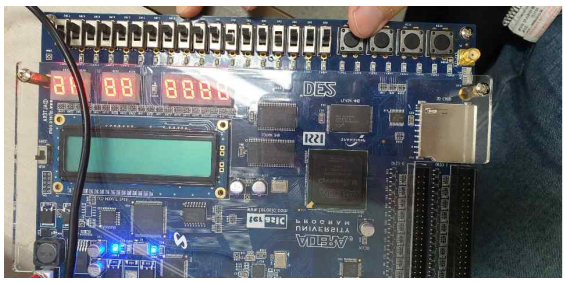
위 아스키 코드표는 A~Z, a~z의 알파벳을 HyperTerminal 에서 어떤 방식으로 보내야 하는지 보여준다.

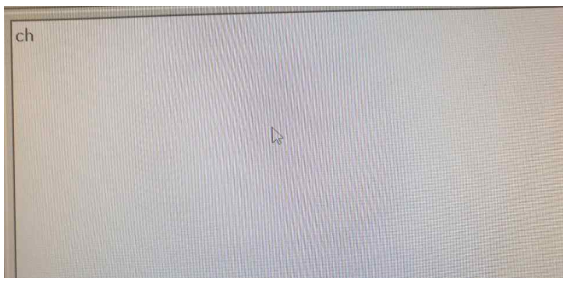
본인은 이름인 CHAN을 전송하려고 하는데, 각 알파벳은
C : 0x45 -> 0b01000011
H : 0x48 -> 0b01001000
A : 0x41 -> 0b01000001
N : 0x4E -> 0b01001110

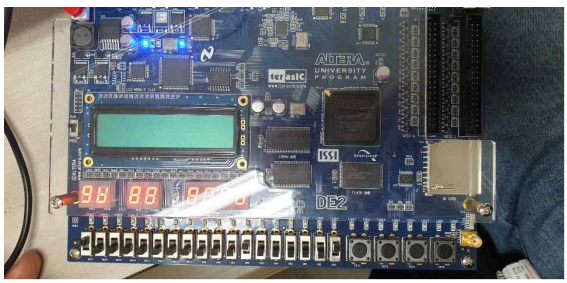
으로 보낼 수 있다.

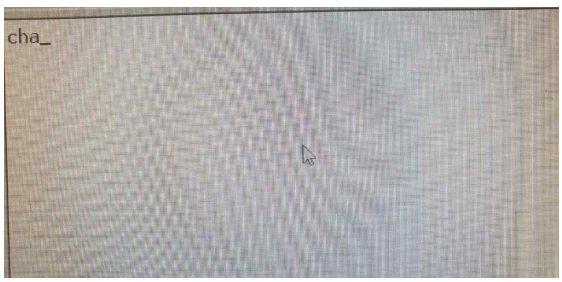


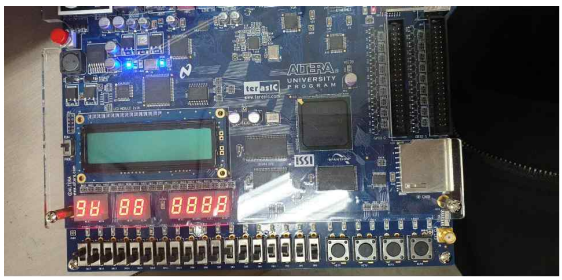


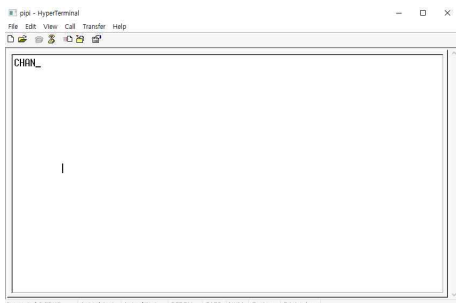








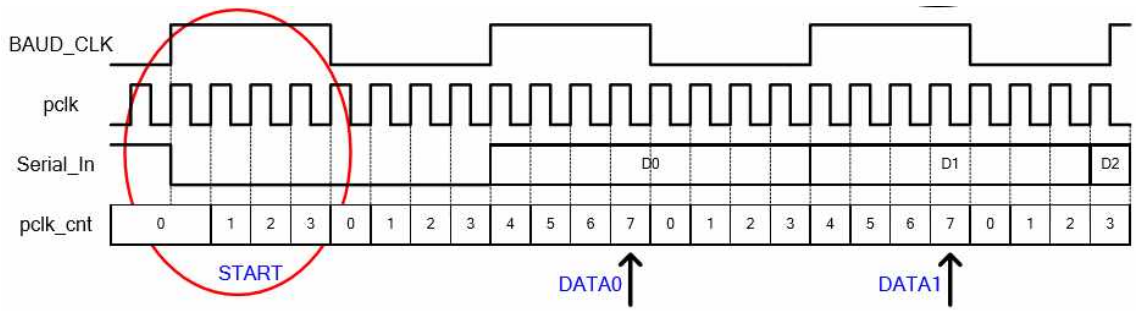


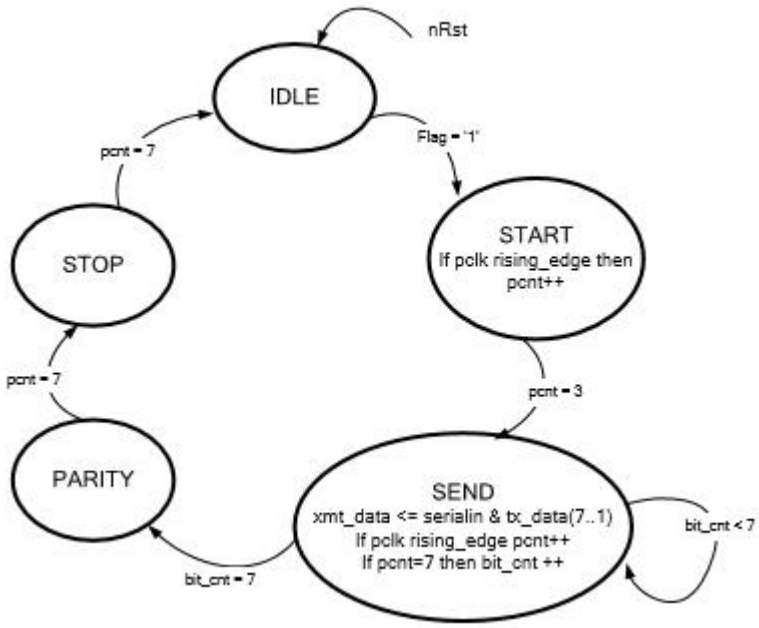


왼쪽의 보드에서 값을 하나씩 보냈을 때 HyperTerminal에서는 아스키 코드가 하나씩 글자로 변환되어 받는 것을 확인할 수 있다.
하나씩 보낸 결과 CHAN의 값을 받은 것을 확인하였다.

UART_RX 설계

• Baud_clk 8분주 클럭을 이용해서 중간위치 값을 취함





다음은 protocol과 Diagram이다.

1) nRst이 '0'일 때, state는 IDLE사태를 유지한다.

2) flag가 '1'이 되면 state는 START로 이동한다. START에서 pcnt의 값은 pclk의 rising_edge에 맞춰 증가한다.

3) pcnt가 3이 되면, state가 SEND로 이동한다. SEND에서 tx_data의 7비트부터 1비트까지 값은 LSB를 제외하고 하나씩 serialin과 &로 연산되어 xmt_data에 저장된다. 위와 마찬가지도 pcnt의 값은 pclk에 맞춰 증가하며 pcnt가 '7'이 되면 bit_cnt를 1증가시킨다.

4) bit_cnt의 값이 '7'이 되면 state는 STOP상태가 된다.

5) pcnt가 7이 되면 state는 IDLE로 되돌아간다.

Code of UART_RX

```
 1    library ieee;
 2       use ieee.std_logic_1164.all;
 3       use ieee.std_logic_arith.all;
 4       use ieee.std_logic_unsigned.all;
 5
 6    entity uart_rx is
 7       port(
 8          nRst     : in std_logic;
 9          clk      : in std_logic; -- clk
10          serialin : in std_logic; -- serial data in
11          rx_data  : out std_logic_vector(7 downto 0);
12          valid    : out std_logic
13       );
14    end uart_rx;
15
16    architecture BEH of uart_rx is
17
18       type   state_type is (IDLE, START, RECEIVE, PARITY, STOP);
19       signal state      : state_type;
20       signal cnt        : std_logic_vector(5 downto 0);
21       signal pclk       : std_logic;
22       signal pcnt       : std_logic_vector(2 downto 0);
23       signal bit_cnt    : std_logic_vector(2 downto 0);
24       signal serialin_d : std_logic;
25       signal flag       : std_logic;
26       signal xmtdata    : std_logic_vector(7 downto 0);
27
```

UART_RX의 library, entity, architecture의 signal선언이다.
위의 TX와 같은 역할을 하므로 따로 설명은 생략하도록 하겠다.

```
28    begin
29
30       process(nRst, clk)
31       begin
32          if(nRst = '0') then
33             cnt  <= (others => '0');
34             pclk <= '0';
35          elsif rising_edge(clk) then
36             if(cnt = 53) then
37                cnt  <= (others => '0');
38                pclk <= not pclk;
39             else
40                cnt  <= cnt + 1;
41             end if;
42          end if;
43       end process;
```

매핑이 끝났으면 process문을 진행시킨다.

8개의 pclk가 868을 8분주하는 값이 되려면 8번의 plck 주기만이 8.68ns의 내에 존재하면 되는데, 값을 구해보면 cnt가 0~53의 변위를 가져 총 54번의 상승 이후 pclk를 '0'과 '1'의 값을 반복하게 만들면 가장 오차가 적은 값을 구할 수 있다.

```
45       process(nRst, clk)
46       begin
47          if(nRst = '0') then
48             serialin_d <= '0';
49             flag       <= '0';
50          elsif rising_edge(clk) then
51             serialin_d <= serialin;
52             if state = IDLE then
53                if(serialin_d = '1')and(serialin = '0') then
54                   flag <= '1';
55                end if;
56             elsif state = START then
57                flag <= '0';
58             end if;
59          end if;
60       end process;
```

nRst의 값이 '0'일 때 serialin_d와 flag는 '0'으로 초기화도니다.
nRst = '1'일 때 clk이 rising_edge이면 serialin_d에 serialin의 값을 할당한다. 이는 clk만큼의 delay가 serailin_d와 serialin의 사이에 존재하게 한다.
다음으로 state가 IDLE상태에서 serialin_d가 '1'이고, serialin이 '0'이면, flag는 '1'을 할당한다. 즉 위의 tx에서 39~40의 값과는 반대로 진행됨을 알 수 있다.
state가 START에 진입하면 flag는 다시 '0'으로 초기화된다.

```
62      process(nRst, pclk)
63      begin
64         if(nRst = '0') then
65            state   <= IDLE;
66            pcnt    <= (others => '0');
67            bit_cnt <= (others => '0');
68            xmtdata <= (others => '0');
69            rx_data <= (others => '0');
70         elsif rising_edge(pclk) then
71
72            case state is
```

다음 process에서 먼저 nRst = '0'이면 state는 IDLE에 위치하고 pcnt, bit_cnt, xmtdata, rx_data의 값들은 모두 초기화된다.
다음 state는 모두 nRst = '1'일 때 pclk의 rising_edge맞추어 확인한다.

```
74               when IDLE =>
75                  if(flag = '1') then
76                     state <= START;
77                  else
78                     state <= IDLE;
79                  end if;
80                  pcnt    <= (others => '0');
81                  bit_cnt <= (others => '0');
82                  xmtdata <= (others => '0');
83                  rx_data <= (others => '0');
```

state가 IDLE일 때 flag가 '1'이면 START로 이동한다.
IDLE에서 pcnt, bit_cnt, xmtdata, rx_data의 값들을 모두 초기화한다.

```
85               when START =>
86                  if(pcnt = 3) then
87                     pcnt  <= (others => '0');
88                     state <= RECEIVE;
89                  else
90                     pcnt  <= pcnt + 1;
91                     state <= START;
92                  end if;
```

state가 START에서 pcnt의 값은 pclk의 rising_edge일 때마다 1씩 상승한다.
이후 pcnt값이 '3'에 도달하면 state는 START에서 RECEIVE모드로 이동하게 된다.

```
94              when RECEIVE =>
95                 if(pcnt = 7) then
96                    pcnt    <= (others => '0');
97                    xmtdata <= serialin & xmtdata(7 downto 1);
98                    if(bit_cnt = 7) then
99                       bit_cnt <= (others => '0');
100                      state   <= PARITY;
101                   else
102                      bit_cnt <= bit_cnt + 1;
103                      state   <= RECEIVE;
104                   end if;
105                else
106                   pcnt <= pcnt + 1;
107                end if;
```

RECEIVE state에서 pcnt가 '7'에 도달할 때 까지 값은 pclk의 rising_edge에 맞추어 증가한다.
그리고 pcnt가 7이 되면 다시 값을 0으로 초기화하고 xmtdata에 serianin과 xmtdata의 7bit부터 1bit까지를 '&'연산하여 할당한다.
그리고 bit_cnt값을 1 증가시킨다.

bit_cnt의 값이 7에 도달하면 state는 RECEIVE로 이동하고 bit_cnt는 0이 된다.

```
109             when PARITY =>
110                if(pcnt = 7) then
111                   pcnt  <= (others => '0');
112                   state <= STOP;
113                else
114                   pcnt  <= pcnt + 1;
115                   state <= PARITY;
116                end if;
117
118             when STOP =>
119                if(pcnt = 7) then
120                   pcnt  <= (others => '0');
121                   state <= IDLE;
122                else
123                   pcnt  <= pcnt + 1;
124                   state <= STOP;
125                end if;
126                rx_data <= xmtdata;
127
128             when others =>
129                state <= IDLE;
```

PARITY, STOP에서 pcnt가 7이 되면 state가 다음으로 진행된다.
STOP에서 rx_data의 값에 xmtdata의 값을 할당한다.

```
128                when others =>
129                   state <= IDLE;
130             end case;
131          end if;
132       end process;
133
134       valid <= '1' when (state = STOP and pcnt = 7) else '0';
135
136    end BEH;
```

valid의 값은 state가 STOP이고 pcnt가 '7'일 때만 '1'의 값을 갖고 나머지는 '0'의 값을 갖는다.

Test_Bench of UART

rx의 경우는 tx와 다르게 tx값이 지정되지 않으면 값을 확인할 수 없다.
그래서 이전에 만들어 둔 UART_TX의 Test Bench에 Rx의 코드를 추가로 작성하여 UART전체에 관한 Test Bench를 만들어 준다.

```
1   library ieee;
2      use ieee.std_logic_1164.all;
3      use ieee.std_logic_arith.all;
4      use ieee.std_logic_unsigned.all;
5
6   entity tb_uart is end;
7
8   architecture BEH of tb_uart is
9
10     component uart_tx is
11        port(
12           nRst      : in std_logic;
13           clk       : in std_logic;
14           start_sig : in std_logic;
15           data      : in std_logic_vector(7 downto 0);
16           tx        : out std_logic;
17           busy      : out std_logic
18        );
19     end component;
20
21     component uart_rx is
22        port(
23           nRst     : in std_logic;
24           clk      : in std_logic; -- clk
25           serialin : in std_logic; -- serial data in
26           rx_data  : out std_logic_vector(7 downto 0);
27           valid    : out std_logic
28        );
29     end component;
30
31     signal nRst      : std_logic;
32     signal clk       : std_logic;
33     signal start_sig : std_logic;
34     signal data      : std_logic_vector(7 downto 0);
35     signal tx        : std_logic;
36     signal busy      : std_logic;
37     signal int_cnt   : std_logic_vector(80 downto 0);
38     signal rx_data   : std_logic_vector(7 downto 0);
39     signal valid     : std_logic;
```

위에 uart_tx의 test bench에 uart_rx의 componen와 signal신호를 추가시킨다.

```
41  begin
42
43      process
44      begin
45          if(NOW = 0 ns) then
46              nRst <= '0', '1' after 200 ns;
47          end if;
48          wait for 1 sec;
49      end process;
50
51      process
52      begin
53          clk <= '0', '1' after 5 ns;
54          wait for 10 ns;
55      end process;
56
57      process(nRst, clk)
58      begin
59          if(nRst = '0') then
60              int_cnt <= (others => '0');
61          elsif rising_edge(clk) then
62              int_cnt <= int_cnt + 1;
63          end if;
64      end process;
```

다음으로 process를 진행시킨다.
처음 process에서는 nRst의 값을 200ns동안 '0'으로 유지시키다 이후로는 계속 '1'로 세트시킨다.

다음 process에서 clk신호는 5ns단위로 '0' '1'값을 번갈아 유지한다.

마지막 process에서는 nRst이 '1'일 때에 int_cnt를 1씩 증가시키고, nRst이 '0'이면 int_cnt의 값을 초기화시키는 과정을 코딩한다.

```
66      start_sig <= '1' when int_cnt = 1000 else
67                   '0';
68      data      <= x"A7" when int_cnt > 900 and int_cnt < 1400 else
69                   x"00";
```

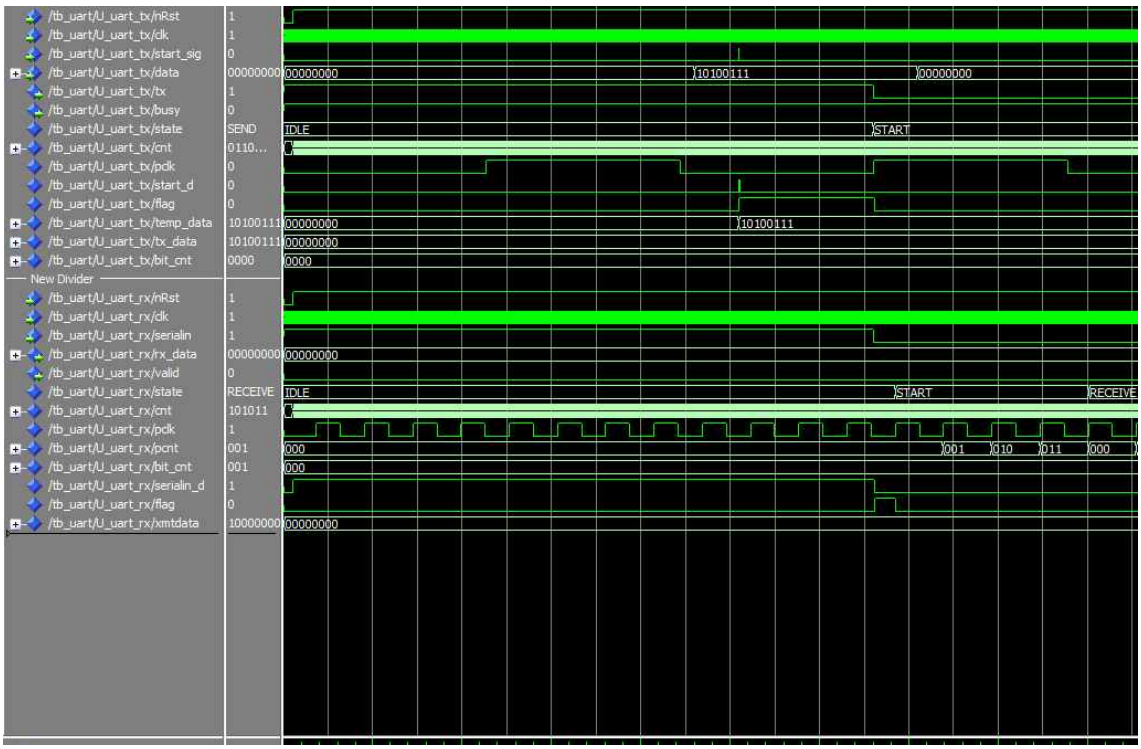
tx와 마찬가지로 세팅하였다.

```
71      U_uart_tx : uart_tx
72      port map(
73         nRst      => nRst,
74         clk       => clk,
75         start_sig => start_sig,
76         data      => data,
77         tx        => tx,
78         busy      => busy
79      );
80
81      U_uart_rx : uart_rx
82      port map(
83         nRst     => nRst,
84         clk      => clk,
85         serialin => tx,
86         --serialin => serialin,
87         rx_data  => rx_data,
88         valid    => valid
89      );
90
91   end BEH;
```

각각의 포트를 라벨링한다.
포트매핑을 완료하고 BEH를 종료한다.

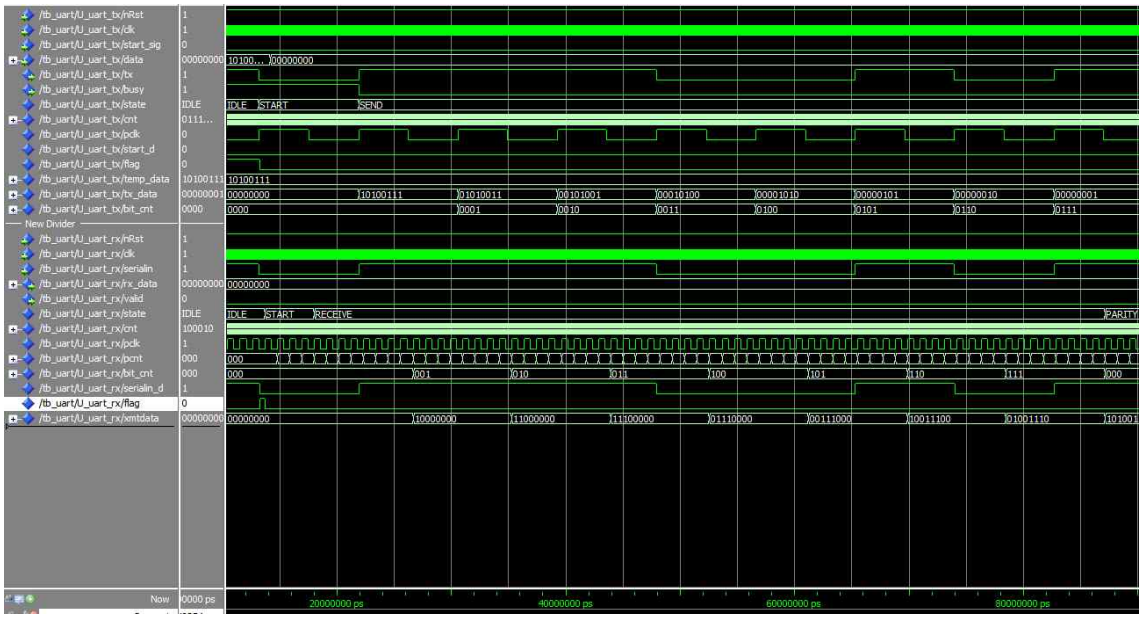
Simulation of UART

다음은 Test Bench를 통해 구현한 UART전체의 시뮬레이션이다.
각 시뮬레이션은 적당한 부분에서 잘라 따로 설명하겠다.



먼저 IDLE상태에서 serialin은 '0', serialin_d는 '1'일 때 flag는 1로 설정된다.
flag는 계속 1로 유지되다가 pclk의 rising_edge에 맞추어 state가 START로 이동하고, 이후 다시 '0'으로 초기화 되는 것을 확인할 수 있다.

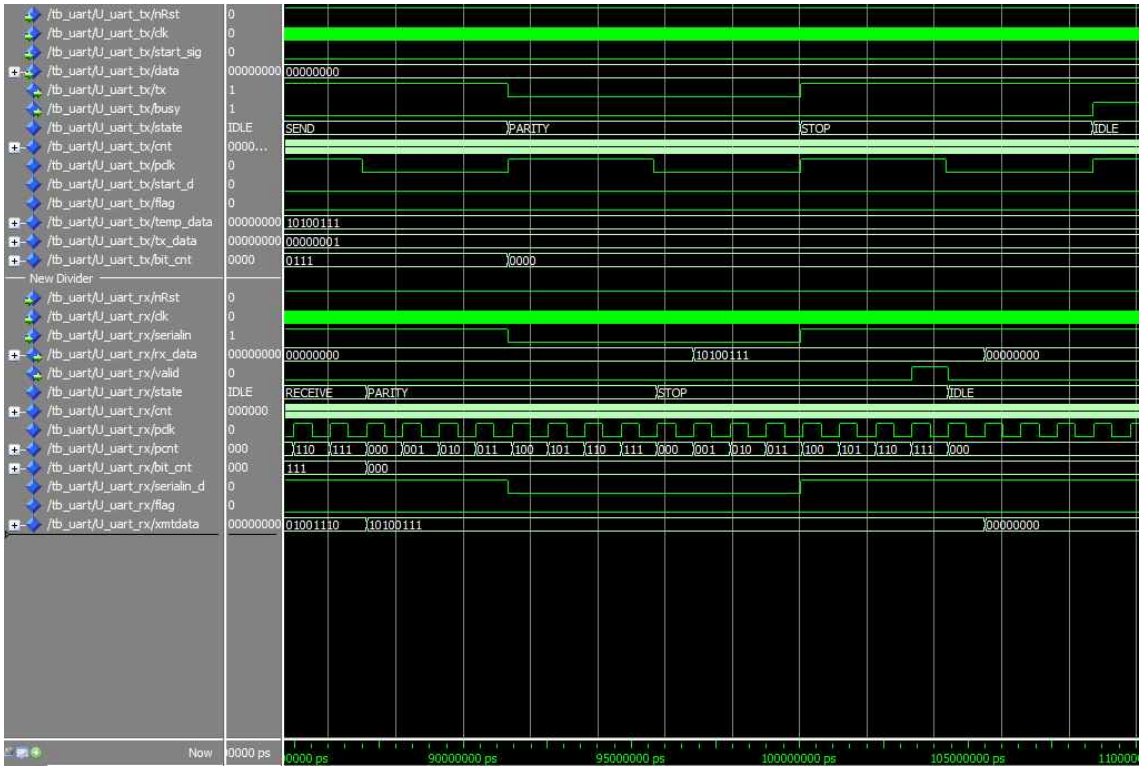
START상태에서 pcnt의 값이 011 즉, 3이 될 때까지 값이 증가하다가 그 값에 도달하면 pcnt가 초기화되고 RECEIVE로 state가 이동함을 볼 수 있다.



RX의 RECIEVE상태가 진행되고 TX는 얼마 뒤 SEND상태로 이동한다.
이는 위 tx의 pclk의 8분주가 rx의 pclk이기 때문에 일어난 현상인데, 정확한 분주가 아니라 둘간에는 약간의 오차가 발생하게 된 것이다.

RX에서 serialin의 값은 TX의 tx_data의 가장 오른쪽 값, 즉 shift가 진행되며 하나씩 밀리는 값이 저장됨이 보인다.
또한 그렇게 받은 serialin값은 본래의 xmtdata의 7번 비트부터 1번 비트까지 각 비트와 '&' 연산되어 xmtdata에 저장되게 된다.
그 값을 확인하여 LSB우선 통신임을 이곳에서 확인할 수 있다.

bit_cnt는 rx의 pclk가 7에 도달할 때마다 1씩 증가하고(크기가 너무 작아 시뮬레이션에서는 pclk값이 표현되지 않았다) 그 bit_cnt가 7에 도달한 이후 state는 PARITY로 이동하고 이와 관련된 pclk와 bit_cnt값은 모두 0으로 초기화된다.



state가 PARITY, STOP을 진행할 때 rx의 pcnt가 3이 될 때마다 다음 state로 넘어가게 된다.

xmtdata의 값은 IDLE로 진입한 후 첫 pclk이 될 때 까지 값을 유지한다.

state가 STOP이고 pclk가 7인 상태에서 valid값은 '1'이 출력된다.

UART전체를 quartus와 hyperterminal을 통해 확인해 본다.

두 가지 시뮬레이션을 진행 할 예정인데,

1. HyperTerminal에서 값을 보내서 그 값을 보드에서 확인하는 시뮬레이션
2. HyperTerminal에서 값을 보내고, 그 값을 다시 HyperTerminal에서 확인하는 시뮬레이션

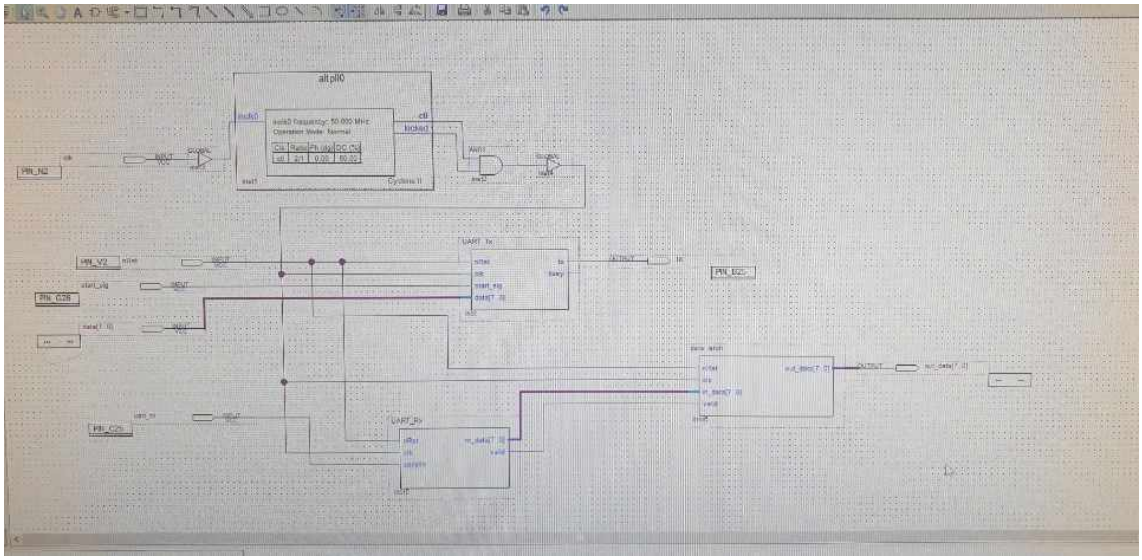
이다.

1) HyperTerminal에서 값을 보내서 그 값을 보드에서 확인하는 시뮬레이션

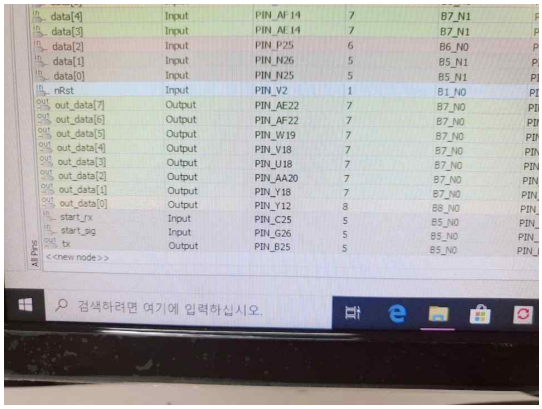
먼저 quartus에서 블록 다이어그램을 만들기 전에 플립플럽을 구성한다.

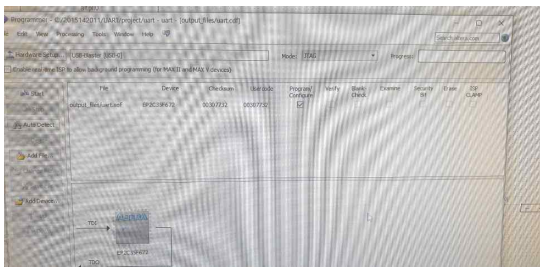
```
library ieee;
	use ieee.std_logic_1164.all;
	use ieee.std_logic_arith.all;
	use ieee.std_logic_unsigned.all;

entity data_latch is
	port(
		nRst		: in std_logic;
		clk		: in std_logic;
		in_data : in std_logic_vector(7 downto 0);
		valid		: in std_logic;
		out_data	: out std_logic_vector(7 downto 0);
	);
end data_latch;

architecture BEH of data_latch is
begin

	process(nRst, clk)
	begin
		if(nRst = '0') then
			out_data <= (others => '0');
		elsif rising_edge(clk) then
			if(valid = '1') then
				out_data <= in_data;
			end if;
		end if;
	end process;

end BEH;
```

다음의 코드를 quartus의 VHDL파일을 만든다.
이 코드를 symbolize하여 블록 다이어그램의 플립플랩으로 사용한다.

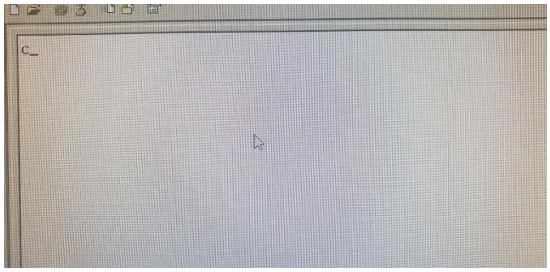


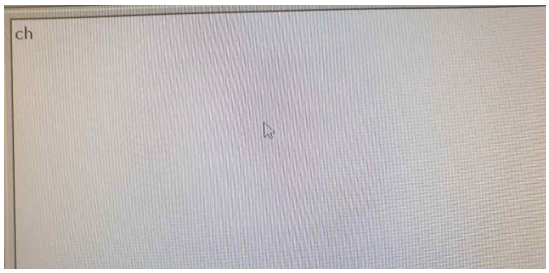
다음은 완성된 블록 다이어그램이다.
UART의 tx, rx와 data_latch의 symbol을 각자 연결하였다.

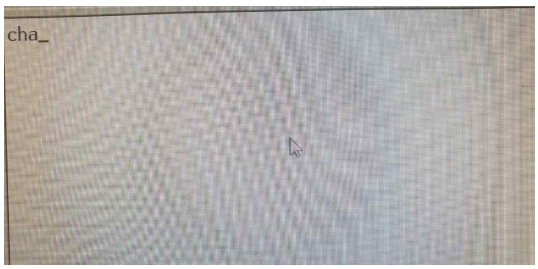


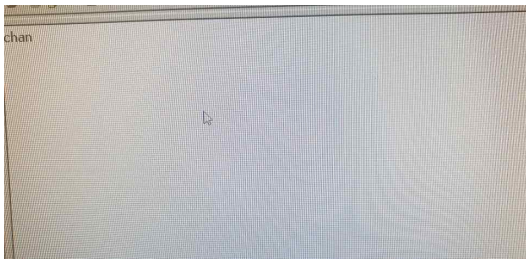


PIN Assignment 진행후 프로그램을 실행한다.
RX는 위에 설정한 대로 값을 기입할 수 있다.



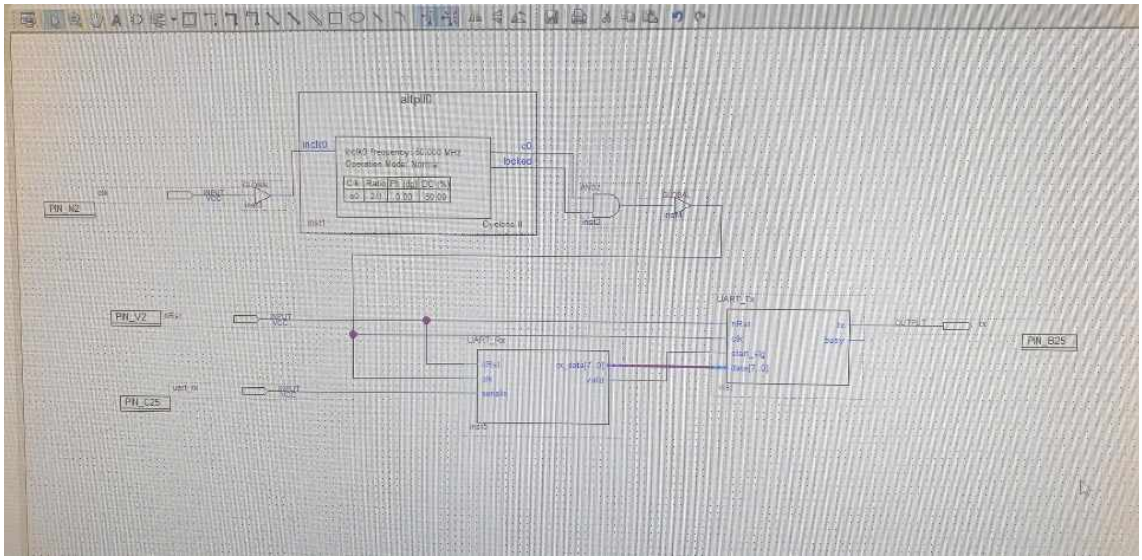


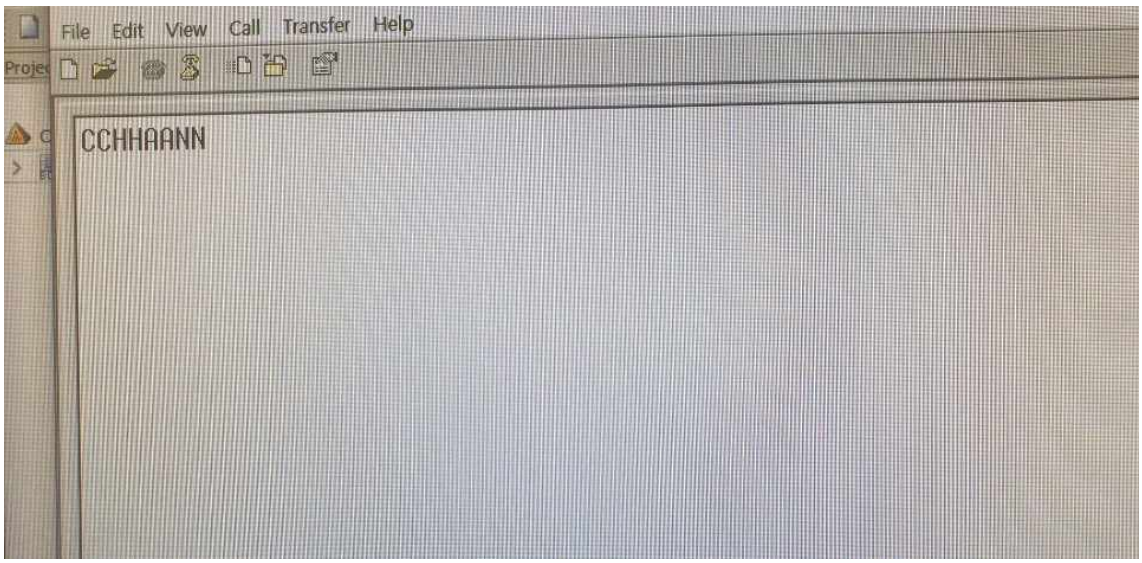




c-h-a-n-의 값을 각각 HyperTerminal에서 보냈을 때 보드에서는 그에 맞는 아스키 코드값을 확인할 수 있다.

2) HyperTerminal에서 값을 보내고, 그 값을 다시 HyperTerminal에서 확인하는 시뮬레이션 (ECHO-BACK simulation)





블록 다이어그램으로 만든 quartus파일을 HyperTerminal로 보내 확인한 값이다.
같은 문자가 두 번씩 찍히는 것을 확인할 수 있다.
-> 자신이 보낸 값을 다시 자신이 받아와서 출력한다. 즉 앞의 글자는 사용자가 입력한 값이고, 동일한 글자가 두 번 나온 것은 입력값을 다시 자동으로 HyperTerminal이 출력시킨 것이다.

결론 및 소감

UART통신을 하기 위해 modelsim과 quartus, HyperTerminal 세 가지 모두를 사용해 보니 SOC라는 과목에 대해 더 세밀하게 공부할 수 있었다.
또한 이전에 레포트로 제출했던 ps2와는 달리 실제로 보드와 HyperTerminal의 동작을 눈으로 확인하며, 원하는 값의 도출을 위해 다른 값을 직접 입력해 볼 수 있어서 훨씬 더 집중도 있게 실습을 진행했던 것 같다.

처음 할 때와는 달리 이번에는 Test_bench의 구현이나, UART의 tx, rx의 코드에 있어서 동작에 필요한 내용들을 빠르게 습득할 수 있었다.

다른 과목에서 마이크로 프로세서2라는 과목이 있는데, 이 과목에서도 UART에 대한 강의를 했었다.

그 때 UART의 전송 특성인 전이중 방식이나 세세한 특성을 공부한 뒤, 이 강의에서 실제 칩 설계에서의 UART를 공부하여, UART라는 방식을 더 잘 알 수 있었던 것 같다.

또 ECHO-BACK의 설계를 하면서 단순히 tx, rx를 통해 컴퓨터와 보드의 통신을 구현하는 것 뿐만이 아니라 컴퓨터-보드-컴퓨터의 통신도 만들 수 있다는 것을 알았다.
그리고 그 ECHO-BACK을 구현하며 방법이 나와있지 않더라도 내가 원하면 ECHO-BACK의 데이터를 컴퓨터에서 출력하면서 동시에 보드에서도 확인 할 수 있게 하는 등 다양한 블록 다이어그램을 만들 수 있을 정도로 그와 관련된 지식이 쌓였다는 것이 느껴졌다.

이것으로 중간고사 전까지의 모든 레포트가 끝났는데, 내가 쓴 레포트와 수업자료, 코드를 보며 시험에 대비할 수 있을 것 같다.